Random Circuit Verilog Code Generation

Tim Price - CSCI 4532

§ · Introduction and Background

1.1 · Circuit Generation

Verilog is a Hardware Description Language which is used to describe the logical behavior of digital circuits. Modules within Verilog define componets or whole circuits, and the goal of this project is to automate the creation of random Verilog modules representing combinational logic circuits.

Each Module created has a common anatomy of gates and wires. Foundational digital logic is created using some combination of the following gates: AND, OR, NAND, NOR, NOT and XOR. Traditionally each gate can have any number of input and output wires refered to as the gate's fan-in and fan-out respectively. For this project all gates that are generated must have a limited fan-in of 2 and can have an unlimited fan-out. This means that any gate must have exactly 2-inputs but can be used to drive inputs for any number of other gates.

To determine the number of internal gates used in the generated circuit, some number of inputs n and some number of outputs m are used to randomly select $\boldsymbol{X}$ number of internal gates in the available range.

$$\boldsymbol{X} \in \{[n+m], [n+m]+1, ..., [\alpha(n+m)]\}$$

$$\text{and } \alpha = x \in \{\mathbb{R}\}.$$

Figure 1: Internal Gate Range

After the total number of gates is selected, all $\boldsymbol{X}$ gates are each randomly assigned a gate type (AND, OR, etc.).

To connect internal nodes, a DAG graph data-structure is used. A *DAG* or directed-acyclic-graph is where each edge or wire has a directed path (can only go in a certain direction) and is acylic meaning there are no loops or cycles in the graph connections which could lead to sequential circuit behavior.

The way to enforce this structure is by using Topological Sorting when assigning each gates inputs. When generating nodes each is put in a list in order of creation.

```
// some code

connect_internal_nodes(...)
{
  for(int i = n; i < internal + n; i++)
  {
    nodes[i].inputA.oid = rand() % i;
    nodes[i].inputB.oid = rand() % i;
    nodes[i].inputA.not = rand() % 2;
    nodes[i].inputB.not = rand() % 2;
  }
}

// some code
```

Listing 1: Internal Node Connection Function

Since the nodes are generated in an ordered list, this loops through each node starting at the first internal node and randomly selecting an internal node or input node that came before it in the list. Additionally, each inputs node's input wire may be inverted to capture the behavior of the NOT gate; NOT's behavior works with single wires instead of requiring a whole gate. This allows any single node's output to be selected by any number of nodes that come after it in the list, fulfilling the unlimited fan-out requirement. To connect nodes to outputs, a random internal node is selected and connected to an output node.

1.2 · Circuit Optimization

It is most likely that there is some redundency in the generated circuit. Not all output nodes are effected by its chain of logic or the preceding inputs.

$$y = a \cdot b \mid a \cdot \hat{b}$$

$$y = a \cdot \left(b \cdot \hat{b}\right)$$

$$y = a$$

Figure 2: Redundant Logic Minimilization

In Figure 2, the output y is independent of its input b and therefore can be removed from the circuit without changing y to reduce or minimize the number of gates and "work" being done by the circuit. This also improves runtime by lowering latency and could reduce cost by removing gates from the resulting circuit.

This process is called logic minimilization or boolean algebra, and is known to have only heuristic solutions for larger expressions. One of the most well known methods of heuristic logic reduction is the "Espresso heuristic logic minimizer." Espresso manipulates binary vectors of an expressions input and output into cube structures then use a set of minimilization algorithms to find an optimal expression. There is no guarantee that this resulting expression will be the global minimimum reduction, but in practice the resulting expression is likely extremely close or completely reduced. It seems to be the most widely accepted solution that works on any size of circuit, so given the unknown number of internal gates generated at runtime, it seems like the best fit for reducing the random circuits.

Espresso accepts input through PLA file types which include $.i$ inputs, $.o$ outputs, rows of input and output vectors and $.e$ the end delimiter.

```
// circuit.pla

   .i 5
   .o 5
   00000 00101
   10000 00101
   01000 00101
   11000 00101
   .e
```

Listing 2: Example PLA Format

Where the binary vectors on the left represent the circuit's input and the columns on the right represent the circuit's output. To optimize the generated circuit the graph must be converted to PLA format which is done by recursively walking backwards from outputs and evaluating each internal node. Then a function writes to a `circuit.pla` file which contains the original unoptimised circuit generated.

`circuit.pla` is then passed into Espresso which then optimizes/minimizes into another `.pla` file, this time in the `./out/` directory. This file, `reduced_circuit.pla`, is the final reduction of the circuits logic.

1.3 · Verilog Generation

To generate a verilog `.v` file I used several python functions. The first function `read_pla` reads the `reduced_circuit.pla` file and makes a struct using its componets.

```
@dataclass
class Circuit:
    i: int
    o: int
    p: int
    io: list [Tuple[str, str]]
```

Listing 3: Reduced PLA File Struct

The second function, `gen_verilog` makes a new file `circuit.v` and sets up a standard verilog module. The generated Verilog module takes in an input vector of size $.i - 1$ and contains an output wire of size $.o - 1$ Each input vector of the PLA file is given a wire with a truth condition set to the combination of the vectors bits. 1 bits add `(x_ == 1)`, 0 bits add `(x_ == 0)` and don't care bits ("-", bits that were minimized and don't effect the circuit) get skipped entirely. Each value is joined with an && and the `x_` is the position of the bit in the vector. For bit in the output wire is set to the combination of all rows whose bit is set to 1 in its corresponding bit position and joined by a logical or. So, `output[1]` would take all rows who have a 1 bit for position 1 in the row.

```
module circuit (
    input wire  ["n":0] in,
    output wire ["m":0] out
);

  wire x0 = in[0];
  // ...
  wire x4 = in["n"];

  wire r0 = (x_ == _) && (x_ == _);
  // ...
  wire r4 = (x_ == _);

  assign out[0] = r_ | r_;
  // ...
  assign out["m"] = r_;

endmodule
```

Listing 4: Verilog File Structure

§ · Moving Forward

2.1 · Use Cases

From this program steps can be taken to create a large dataset of verilog modules. A large database of Verilog modules could be used to benchmark verification tools. Every tool should return the same output if a testbench was generated alongside the `circuit.v` file. So, for someone developing a new Verilog simulator, having a large amount of variety in source to test against can help uncover subtle inconsistencies. Another idea would be to keep a database of both minimized and un-minimized circuits so that comparisons could be drawn from the minimization. The paired circuits could be used for training machine learning models to recognize what structures in a circuit are most likely going to be minimized.

2.2 · Future Program Additions

Since each module should be synthesizable test bench isn't totally necissary. But if I were to generate verilog for non-optimized circuits, also generating a testbench and comparing the values would be helpful.

Adding combinational logic support would make the generated circuits more interesting and useful.

§ · Contributions

I chose to work on my project alone, so all code outside of the Espresso minimizer was written by myself. Everything in `gen_verilog.py` and `circuit.c` as well as the `makefile` was all developed by me.

A very large contribution to my project was the Espresso Heuristic Logic Minimizer which was originally developed by "Brayton et al. at the University of California, Berkeley." The implementation I used within my system was written by the github user hadipourh and is linked in the references section of this paper as well as the README in my source code.

§ · Challenges

The biggest challenge I faced while developing my circuit generator was minimizing the logic. I believe unrestricted boolean logic minimalization falls into the category of NP-complete problems. I spent lots of time researching possible solutions that I could reimplement, but the complexity of this problem is simply out of my current paygrade. So, while I'm not thrilled with building off the open source Espresso repository I used, it does seem like the best solution given the constraints of my project.

Another issue is the runtime after compilation, and the memory usage during circuit generation. Increasing my alpha value much higher than 10 results in the program crashing on my laptop which I can only assume is a memory overflow when it does work on my desktop. The runtime problem persist both with my circuit generation and with the Espresso minimizer. It takes multiple minutes for the program to generate and minimize larger sized circuits. The highest I went when testing was `n = 20` and `m = 20` and an alpha of 10 which seems to work but needs to be better optimized. Its worth mentioning that I'm not a very good C programmer and likely have memory issues causing unnecissary overhead. It does seem like generating very large circuits is really quite difficult due to memory constraints and future implementation might benefit from some technique to store the state of the graph in something other than memory and generating it in parts.

I spent some amount of time trying to visualise the logic graph generated by the circuit but that also resulted in a dead-end. I imagined something like the unix tree command that shows how directories link together but with the nodes and wires from the logic graph. I think it would be interesting to see how different the structures look between random generations but ultimately didn't have time to spend on that extra step.

I found converting PLA files to Verilog to be initially quite tricky. I originally intended to write everything in C but decided that for writing text to a file python was the superior option. While the python program is really not well written, it does work and it is reasonably fast.

§ · Deliverables

The source deliverable folder contains a `circuit.c` and `gen_verilog` files. The Espresso Heuristic Logic Minimizer is contained in the `Expresso` folder which has all of its non-compiled source code.

To compile and run the entire system a makefile is inclduded within the directory. Simply run the `make` command and all source files should build in order, then run accordingly. Lots of output will be echoed to visualize the stage of the program because it can take several minutes to run at larger sizes If something were to go wrong during compilation, or the user should wish to delete the programs output and executables, the `make clean` command will delete the contents of the `./out/` directory, clean the compiled Espresso program, and remove the executable files generated in the source directory. There should be a "cleaned" message returned if the operation was successful.

All of the important output should be found within the `./out/` directory. This should contain both `circuit.v` and `reduced_circuit.pla`. Should you want to view the non-minimized pla file, it can be found after compilation in the source directory.

A full list of dependencies can be found in the "Setup and Results section"

§ · Setup and Results

6.1 System Requirements

A linux based operating system is recommended for running my program. I have only tested it on "Fedora Linux 41." I would assume since all the depencies are cross platform it should work on other systems, but do disclaim the lack of testing on other systems.

I have run my program on both my laptop and desktop with much better results coming from my desktop. Due to memory constraints on my laptop I have run into several program crashes when generating larger circuits, and my 32gb of ram on my desktop has been, not fully, but more reliable. I also get my output returned quite a bit quicker. Most of my good testing was done with a "AMD Ryzen 5 5600." The program is still painfully slow at larger sizes but it is a level of magniture quicker on the desktop. I would highly recommend using a desktop system when generating large sized circuits.

6.2 · Dependencies

this list contains depencies and the versions that were used whilst testing and developing

- gcc - 14.3.1
- python3 - 3.13.9
- make - 4.4.1
- GNU bash - 5.2.32
- hadipourh/espresso - 3.0 (included in src)
- steveicarus/iverilog - 11.0+ (optional)

other versions may work, but have not been tested.

6.3 · Generating Circuits of Different Size

As mentioned in the deliverables section, I hope the makefile included should make it easy to compile and run the source code. There currently is no slick way to change the size of the circuit. The way to change the generation parameters is modifying the main method in the `circuit.c` file.

```c
int main() {
    srand(time(NULL));
    int n = 20; int m = 20;
    int internal = 0;
    int total_nodes = numInternal(...);
    Node* nodes = genNodes(...);

    tableGen(nodes, n, m, internal);
    return 0;
```

Listing 5: Circuit Generation Parameters

In this function, `int n = ...` and `int m = ...` are the input and output node counts. The internal will be generated at runtime and should remain zero. In the `numInternal()` function call, an integer is accepted after n and m, and that is the α value referenced in section 1.1. Making this number higher than 20 will likely result in crashes, but will need to be tested on a machine-by-machine basis.

6.4 · Observations

My main takeaway from my output is the comparison in size from the initial generated circuit and the reduced circuit. When generating a circuit with parameters $n = 20$, $m = 20$ and $\alpha = 10$ the size is quite large, and the number of initial internal gates is printed to the console during the generation process. The initial `circuit.pla` file is multiple thousands of lines long, and the reduced circuit is hardly a fraction. I really want to generate what the initial verilog file would look like just for scale comparison.

§ · References

- Wikipedia (12/1/2025). Directed Acyclic Graph. Wikipedia. https://en.wikipedia.org/wiki/Directed_acyclic_graph
- Wikipedia (11/23/2025). Topological Sorting. Wikipedia. https://en.wikipedia.org/wiki/Topological_sorting
- Wikipedia (10/20/2025). Logic Optimization. Wikipedia. https://en.wikipedia.org/wiki/Logic_optimization
- Wikipedia (6/30/2025). Espresso heuristic logic minimizer. Wikipedia. https://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer
- Hadipourh (10/27/2025). espresso. https://github.com/hadipourh/espresso
- Richard L. Rudell (Jun 1986). Multiple-Valued Logic Minimization for PLA Synthesis. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1986/ERL-86-65.pdf
- Peeter Ellervee () Espresso Explained, How espresso works and what is behind this. https://www.tud.ttu.ee/im/Peeter.Ellervee/IAS0540/espresso-explained.pdf
- University of California Berkley (), Espresso, a Multi-valued PLA minimization. https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm